P-Ring: An Index Structure for Peer-to-Peer Systems

Adina Crainiceanu Prakash Linga Ashwin Machanavajjhala
Johannes Gehrke Jayavel Shanmugasundaram
Department of Computer Science, Cornell University
{adina,linga,mvnak,johannes,jai}@cs.cornell.edu

Abstract

Current peer-to-peer (P2P) index structures only support a subset of the desired functionality for P2P database systems. For instance, some P2P index structures support equality queries but not range queries, while others support range queries, but do not support multiple data items per peer or provide guaranteed search performance. In this paper, we devise a novel index structure called P-Ring that supports both equality and range queries, is fault-tolerant, provides guaranteed search performance, and efficiently supports large sets of data items per peer. We are not aware of any other existing index structure that supports all of the above functionality in a dynamic P2P environment. In a thorough experimental study we evaluate the performance of P-Ring and quantify the performance trade-offs of the different system components. We also compare P-Ring with two other P2P index structures, Skip Graphs and Chord.

1. Introduction

Peer-to-peer (P2P) systems are emerging as a new paradigm for structuring large-scale distributed systems. The key advantages of P2P systems are their scalability, due to resource-sharing among cooperating peers, their fault-tolerance, due to the symmetrical nature of peers, and their robustness, due to self-reorganization after failures. Due to the above advantages, P2P systems have made inroads for content distribution and service discovery applications [1, 33, 29, 30]. However, most existing systems only support location of services based on their name, i.e., they only support equality queries.

In this paper, we argue for a much richer query semantics for P2P systems. We envision a future where users will use their local servers to offer services described by semantically-rich XML documents. Users can then *query* this "P2P service directory" as if all the services were registered in one huge centralized database. As a first step towards this goal we propose P-Ring, a new distributed fault-tolerant index structure that can efficiently support *range queries* in addition to equality queries. P-Ring is fault-tolerant, gives guaranteed logarithmic search performance in a consistent system, and supports possibly large sets of items per peer. Such an index structure could be used by sophisticated P2P database applications such as digital libraries [22]. We are not aware of any other existing index structure that supports all of the above functionality in a dynamic P2P environment.

When designing P-Ring we were faced with two challenges. First, we had to distribute data items among peers in a such a way that range queries could be answered efficiently, while still ensuring that all peers had roughly the same number of data items (for storage balance). Existing techniques developed for equality queries are not applicable in our scenario because they distribute data items based on their hash value; since hashing destroys the order of the data items, range queries cannot be answered efficiently. We thus need to devise a scheme that clusters data items by their data *value*, and balances the number of data items per peer even in the presence of highly skewed insertions and deletions. Our first contribution is a scheme that provably maintains a maximum load imbalance factor of at most $2 + \epsilon$ between any two peers in the system, while achieving amortized constant cost per insertion and deletion.

Our second challenge was to devise a query router that is robust to failures and provides logarithmic search performance even in the presence of highly skewed data distributions. Our P-Ring router is highly fault-tolerant, and a router of order d provides guaranteed $O(\log_d(P))$ search performance in a stable system with P peers. Even in the presence of highly skewed insertions, we can guarantee a worst-case search cost of $O(x \cdot d \cdot \log_d(P))$, where x is the number of insertions per stabilization unit of the router (we will formally define all terms later in the paper).

In a simulation study, we compare the performance of PRing to an extension of SkipGraphs [2], the only other P2P router that we are aware of that provides provable search guarantees for range queries over arbitrary ordered domains. Our performance results indicate that P-Ring outperforms the above extension of Skip Graphs in terms of both query and update cost. Surprisingly, P-Ring sometimes outperforms Chord, an index structure designed for equality queries, even in the case of equality queries.

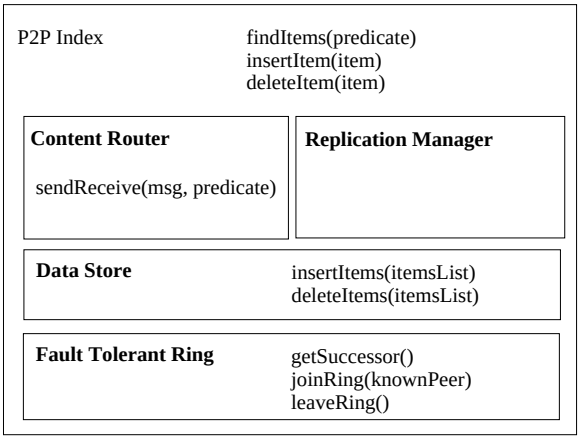

Figure 1. Indexing Framework

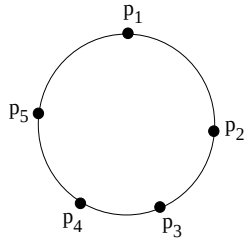

Figure 2. Fault Tolerant Ring

2. System Model and Architecture

2.1. System Model

A *peer* is a processor that has some shared storage space and some private storage space. The shared space is used to store the distributed data structure for speeding up the evaluation of user queries. We assume that each peer can be identified by a physical id, which can be its IP address. We also assume a fail-stop model for peer failures. A *P2P system* is a collection of peers. We assume there is some underlying network protocol that can be used to send messages reliably from one peer to another with bounded delay. A peer can join a P2P system by contacting some peer that is already part of the system. A peer can leave the system at any time without contacting any other peer.

We assume that each data item stored in a peer exposes a *search key value* from a totally ordered domain that is indexed by the system. Without loss of generality, we assume that search key values are unique (duplicate values can be made unique by appending the physical id of the peer where the value originates and a version number; this transformation is transparent to users). Peers inserting data items into the system can retain the ownership of their items. In this case, the data items are stored in the private storage partition at the peer and only pointers to the data items are inserted into the system. In the rest of the paper we make no distinction between data items and pointers to the data items.

2.2. System Architecture

We have implemented P-Ring in the context of the PEPPER system [5], which provides a modular framework for implementing new P2P index structures (Figure 1). We now describe the relevant components of the framework.

Fault Tolerant Ring: The Fault Tolerant Ring connects the peers in the system along a ring, and provides reliable connectivity among these peers even in the face of peer failures. For a peer p, we can define the succ(p) (respectively, pred(p)) to be the peer adjacent to p in a clockwise (resp., counter-clockwise) traversal of the ring. Figure 2 shows an example of a Fault Tolerant Ring. If peer p_1 fails, then the ring will reorganize such that succ(p_5) $= p_2$, so the peers remain connected. Figure 1 shows the ring API. When invoked on a peer p, `p.getSuccessor` returns the address of succ(p). `p.joinRing(knownPeer)` inserts p into an existing ring by contacting `knownPeer`. `p.leaveRing` allows p to gracefully leave the ring (of course, p can leave the ring without calling `leaveRing` due to a failure). In our implementation of P-Ring, we use Chord's Fault Tolerant Ring [33].

Data Store: The Data Store is responsible for distributing data items to peers. Ideally, the distribution should be uniform so that each peer stores about the same number of items, thereby achieving storage balance. The Data Store provides API methods to insert and delete items into the system. One of the main contributions of this paper is a new Data Store for P-Ring, which can effectively distribute data items even under highly skewed insertions and deletions (see Section 3).

Content Router: The Content Router is responsible for efficiently routing messages to peers that contain data items satisfying a given predicate. The second major contribution of this paper is a new Content Router that can route range queries efficiently (see Section 4).

Replication Manager: The Replication Manager ensures that items assigned to a peer are not lost if that peer fails. For P-Ring, we use the Replication Manager proposed in [7].

P2P Index: The P2P Index is the index exposed to the end user. It supports search functionality by using the functionality of the Content Router, and supports item insertion and deletion by using the functionality of the Data Store.

3. P-Ring Data Store

One of the main challenges in devising a Data Store for P2P range indices is handling data skew. Ideally we would like the data items to be uniformly distributed among the peers so that the storage load is nearly evenly distributed among the peers. Most existing P2P index structures achieve this goal by hashing. Data entries are assigned

to peers based on the hash value of their search key. Such an assignment has been shown to be very close to a uniform distribution of entries with high probability [29, 30, 33, 36]. However, hashing destroys the value ordering among the search key values, and thus cannot be used to process range queries efficiently (for the same reason that hash indices cannot be used to handle range queries efficiently).

Since P-Ring is designed to support range queries, we assign data items to peers directly based on their search key value. In this case, the ring ordering is the same as the search key value ordering wrapped around the highest value. The problem is that now, even in a stable P2P system with no peers joining or leaving, some peers might become highly overloaded due to skewed data insertions and/or deletions. We need a way to dynamically reassign and maintain the ranges associated to the peers. The next section presents our algorithms for handling data skew. In concurrent work, Ganesan et al. [11] also propose a load balancing scheme for data items where they prove a bound of 4.24 for storage imbalance with constant amortized insertion and deletion cost. Our P-Ring Data Store achieves a better storage balance factor of $(2+\epsilon)$ with the same amortized cost for insertions and deletions.

3.1. Handling Data Skew

The search key space is ordered on a ring, wrapping around the highest value. The Data Store partitions this ring space into ranges and assigns each of these ranges to a different peer. Let $p.range = (p.l, p.u]$ denote the range assigned to p. All the data entries in the system whose search key lies in $p.range$ are said to be $owned$ by p. Let $p.own$ denote the list of all these entries. Let $|p.range|$ denote the number of entries in $p.range$ and hence in $p.own$. The number of ranges is less than the total number of peers in the system and hence there are some peers which are not assigned any ranges. Such peers are called *free peers*. Let $p.ringNode$ refer to the Fault Tolerant Ring component of the P-Ring at peer p.

Analogous to B+-tree leaf page maintenance, the number of data entries in every range is maintained between bounds $lb = \texttt{sf}$ and $ub = 2 \cdot \texttt{sf}$,[1] where `sf` is the "storage factor", a parameter we will talk more about in Section 3.2. Whenever the number of entries in p's Data Store becomes larger than ub (due to many insertions into $p.range$), we say that an *overflow* occurred. In this case, p tries to *split* its assigned range (and implicitly its entries) with a free peer. Whenever the number of entries in p's Data Store becomes smaller than $lb = \texttt{sf}$ (due to deletions from $p.range$ is responsible for), we say that an *underflow* occurred. Peer p tries to acquire a larger range and more entries from its successor in the ring. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a *free peer*. We propose an extension to this basic scheme in Section 3.3, where we use the free peers in the system to help balance the load amongst all the peers such that the ratio between the load on the most loaded peer to the load on the least loaded peer is bounded by a small constant.

1 A factor larger than 2 for the overflow condition is used in the extension to this scheme proposed in Section 3.3

Algorithm 1 : p.`split()`

```
1: p' = getFreePeer();
2: if p' == null then
3:    return;
4: end if
5: //execute the split
6: splitItems = p.own.splitSecondHalf();
7: splitValue = splitItems[0];
8: splitRange = p.range.splitLast(splitValue);
9: p'::joinRingMsgHandler(p,splitItems,splitRange);
```

Algorithm 2 : p'.`joinRingMsgHandler`(p, $splitItems$, $splitRange$)

```
1: p'.range = splitRange;
2: p'.own = splitItems;
3: p'.ringNode.joinRing(p);
```

An Example. Consider the Data Store in Figure 3 which shows the free peers (p_6 and p_7), and the ranges and key values of entries assigned to the other peers in the system (range $(5, 10]$ with data entries with search keys 6 and 8 are assigned to peer p_1 etc.). Assume that `sf` is 1, so each peer in the ring can have 1 or 2 entries. When a data entry with search key 9 is inserted into the system, it will be stored at p_1, leading to an overflow. As shown in Figure 4, the range $(5, 10]$ is split between p_1 and the free peer p_6. p_6 becomes the successor of p_1 on the ring and p_6 is assigned the range $(6, 10]$ with data entries with search keys 8 and 9.

Split. Algorithm 1 shows the pseudo-code of the *split* algorithm executed by a peer p that overflows. We use the notation $p::fn()$ when function $fn()$ is invoked at peer p. During a split, peer p tries to find a free peer p' and transfer half of its items (and the corresponding range) to p'. (The details of how a free peer is found are given in the next section.) After p' is found (line 1), half of the entries are removed from $p.own$ and $p.range$ is split accordingly. Peer p then invites the free peer p' to join the ring as its successor and maintain $p'.range$. The main steps of the algorithm executed by the free peer p' are shown in Algorithm 2. Using the information received from p, p' initializes its Data Store component, the Ring component and the other index components above the Data Store.

Merge and Redistribution. If there is an underflow at peer p, p executes the merge algorithm given in Algorithm 3. Peer p invokes the `initiateMergeMsgHandler`

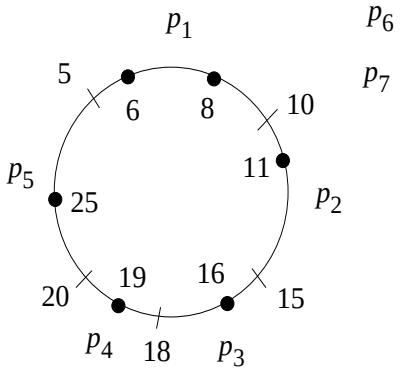

Figure 3. Data Store

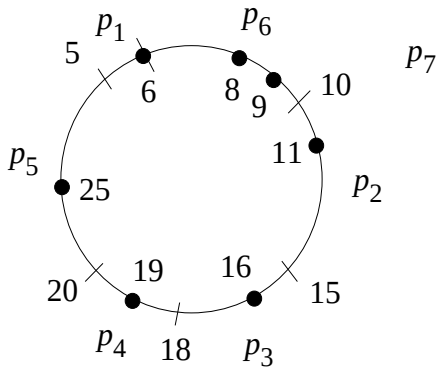

Figure 4. Data Store After Split

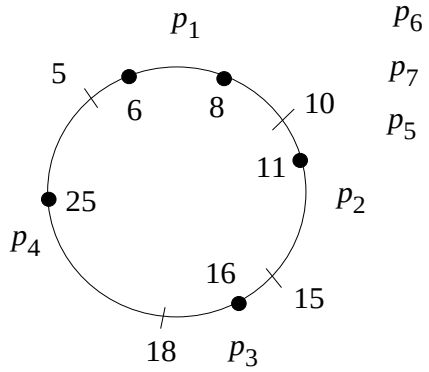

Figure 5. Data Store After Merge

Algorithm 3 : p.`merge()`

```
1: //send message to successor and wait for result
2: (action, newRange, newItemsList) =
   p.ringNode.getSuccessor()::
   initiateMergeMsgHandler(p, |p.range|);
3: p.list.add(newItemsList);
4: p.range.add(newRange);
```

Algorithm 4 : $(action, newRange, newItemsList)$ p'.`initiateMergeMsgHandler`(p,$numItems$)

```
1: if numItems + p'.size > 2 · sf then
2:    //redistribute
3:    compute nbItemsToGive;
4:    splitItems = p'.list.splitFirst(nbItemsToGive);
5:    splitValue = splitItems.lastValue();
6:    splitRange = p'.range.splitFirst(splitValue);
7:    return (redistribute,splitRange,splitItems);
8: else
9:    //merge and leave the ring
10:   splitItems = p'.list;
11:   splitInterval = p'.range;
12:   p'.ringNode.leaveRing();
13:   return (merge, splitRange, splitItems);
14: end if
```

function on its successor on the ring. The successor sends back the action decided, `merge` or `redistribute`, a new range $newRange$ and the list of entries $newItemsList$ that are to be re-assigned to peer p (line 2). p appends $newRange$ to $p.range$ and $newItemsList$ to $p.own$.

The outline of the `initiateMergeMsgHandler` function is given in Algorithm 4. The invoked peer, $p' = \texttt{succ}(p)$, checks whether a redistribution of entries is possible between the two "siblings" (line 1). If yes, it sends some of its entries and the corresponding range to p. If a redistribution is not possible, p' gives up all its items and its range to p', and becomes a free peer.

Example Let us consider again Figure 3 and assume that item with search key value 19 is deleted from the system. In this case, there is an underflow at peer p_4 and peer p_4 calls `initiateMergeMsgHandler` in p_5. Since p_5 has only one item, redistribution is not possible. Peer p_5 sends its data entry to p_4 and becomes free. As shown in Figure 5, peer p_4 now owns the entries in the whole range $(18, 5]$.

3.2. Managing Free Peers

Recall that free peers are used during splits and are generated during merge. There are three important issues to be addressed when managing free peers. First, we should have a reliable way of "storing" and finding free peers. Second, we need to ensure that a free peer exists when it is needed during split. Finally, even though free peers do not have a position on the ring, they are a part of the system and should be able to query the data in the system.

To solve the first issue, we create an artificial entry $(\perp, p'.physicalId)$ for every free peer p' , where $\perp$ is the smallest possible search key value. This artificial entry is inserted into the system like any regular entry. Using this implementation, storing or removing a free peer is similar to inserting or respectively removing a data item from the P2P system. When a free peer is needed, an equality search for $\perp$ is issued. This search is processed as a regular user query and the result is returned to the peer issuing the request.

To ensure that a free peer exists when needed during split, we employ the following scheme: let N be the number of entries in the system and n be the number of peers in the system. If we set `sf`$= \lceil N/n \rceil$, a free peer is guaranteed to exist in the system any time an overflow occurs. `sf` can either be estimated in a very conservative way so that a free peer always exists when needed, or can be adjusted from time to time by estimating N and n using background gossip style aggregation, say like in [19].

To solve the final issue, each free peer maintains a list of non-free peers so that it can forward any query it receives to one of the non-free peers to be processed.

3.3. Load Balancing using Free Peers

Though the scheme proposed in the previous sections maintains the number of entries owned by each peer within strict bounds, there are some peers who do not store any of the data entries. So in a true sense there is no "load balance" amongst the peers. In this section we propose an extension to the basic scheme, which uses the free peers to help "truly" balance the load on the peers. The extended scheme proposed is provably efficient, i.e., every insert and delete of a data entry has an amortized constant cost. Also the *load imbalance*, defined as the ratio of the load on the most loaded peer to the load on the least loaded peer, is bounded by a small constant. We inform the reader that the system evaluated in the experiments does not implement these extensions to the basic scheme.

As a first step toward the algorithm, observe that if we assign data entries to free peers too in some clever way while maintaining the strict bounds lb and ub on the number of entries assigned, we should be able to bound the load imbalance by the ratio $\frac{ub}{lb}$. Inspired by this observation, we introduce the concept of "helper peers". Every free peer is obliged to "help" a peer p already on the ring. The free peer helps p by managing some part of $p.range$ and hence some of the data entries owned by p. If p has k helpers $q_1, q_2, \ldots, q_k$, $p.range = (l, u]$ is divided into $(l = b_0, b_1], (b_1, b_2], \ldots, (b_k, u]$ such that each of the ranges has equal number of entries. Peer p is now *responsible* for $(b_k, u]$. Each of p's helpers, q_j, becomes *responsible* for one of the other ranges, say $(b_{j-1}, b_j]$. Call the list of entries peer q is *responsible* for to be $q.resp$ and call the corresponding range $q.range_{resp}$. All queries dealing with $q.range_{resp}$ will reach q. However, these helpers do not own any entries. Any inserted or delete that reaches q is forwarded to the peer p which actually owns the range and p will see to that all the entries it owns are always evenly divided amongst the helpers. p is also the one that initiates the load balancing algorithm, based on the entries in $(l, u]$. Note that if a non free peer has no helpers, $p.range_{resp} = p.range$, and $p.resp = p.own$.

Let us set up some notation. Consider a set of n peers $\mathcal{P}$. Consider a ring ordered key space and some multi-set of key values on it. Consider a partition $\mathcal{R}$ of the ring, $|\mathcal{R}| < n$, such that $\forall (l, u] \in \mathcal{R}$, $lb \leq |(l, u]| \leq ub$, $\frac{ub}{lb} \geq 2$. Let $\rho : \mathcal{R} \rightarrow \mathcal{P}$ be a 1-1 map defining the assignment of ranges to peers. For every $(l, u] \in \mathcal{R}$, $p = \rho((l, u])$ implies $p.range = (l, u]$ and $p.own$ is the set of entries which lie in $(l, u]$. We can redefine the `succ` and `pred` of p in terms of the ranges in $\mathcal{R}$ and ρ as follows: $p_1 = \texttt{succ}(p)$ if $p = \rho((l, u])$ and $p_1 = \rho((l_1, u_1])$ and $l_1 = u$. Similarly, $p_2 = \texttt{pred}(p)$ if $p = \rho((l, u])$ and $p_2 = \rho((l_2, u_2])$ and $u_2 = l$.

The set $\mathcal{N} = (\rho(\mathcal{R}))$ is the set of non free peers in the system. The set $\mathcal{F} = \mathcal{P} \setminus \mathcal{N}$ is the set of free peers. Let $\psi : \mathcal{F} \rightarrow \mathcal{N}$ be a function defining the helper peer assignments. For a non free peer $p \in \mathcal{N}$, let $\mathcal{H}(p) = \{q | q \in \mathcal{F}, \psi(q) = p\}$; i.e., $\mathcal{H}(p)$ is the set of free peers assigned to p. Note that $\mathcal{R}$ and ρ completely define the sets $\mathcal{N}$ and $\mathcal{F}$ and also define which non free peer $p \in \mathcal{N}$ *owns* each range on the ring space (thus defining $p.own$). Also, for every peer $q \in \mathcal{P}$, $\mathcal{R}$ and ρ coupled with ψ completely define which range q *responsible* for (thus defining $q.resp$). Note that though free peers are responsible for ranges on the ring, the successor and predecessor of a non free peer are still defined in terms of the ownership, i.e., as defined in terms of $\mathcal{R}$ and ρ. We call the tuple $(\mathcal{R}, \rho, \psi)$ as a *configuration* of the system data store.

Definition 1 (Load Imbalance) *Consider a configuration of the data store $(\mathcal{R}, \rho, \psi)$. This completely defines the own and $resp$ sets for each peer $p \in \mathcal{P}$. We define the* load imbalance *as*

$$\frac{\max_{p \in \mathcal{P}} |p.resp|}{\min_{p \in \mathcal{P}} |p.resp|}$$

The extended scheme, which we call Algorithm EXTLOADBALANCE has three load balancing operations:

Split Operation:

CONDITION: A non free peer p *splits* when $|p.own| \geq ub$; i.e., the number of entries owned by p reached the upper bound ub. This operation requires the existence of a free peer, which is guaranteed by Lemma 1.

ALGORITHM:

1: **if** $|p.own| < ub$ **then**
2: return;
3: **end if**
4: //execute the split
5: **if** $\mathcal{H}(p) == \emptyset$ **then**
6: $q = findFreePeer()$;
7: $p.setHelperPeer(q)$;
8: **else**
9: q = some peer in $\mathcal{H}(p)$;
10: **end if**
11: //now p has at least one helper peer q
12: $splitItems = p.own$.`splitSecondHalf()`;
13: $splitValue = splitItems[0]$;
14: $splitRange = p.range$.`splitLast(`$splitValue$`)`;
15: q::`joinRingMsgHandler(`p,$splitItems$,$splitRange$`)`;
16: **if** $\mathcal{H}(p) \setminus \{q\}$!= $\emptyset$ **then**
17: transfer half the helpers from p to q to get $\mathcal{H}'(p)$ and $\mathcal{H}'(q)$;
18: **end if**
19: redistribute $p.own$ amongst $\mathcal{H}'(p)$;
20: rdistribute $q.own$ amongst $\mathcal{H}'(q)$;

PURPOSE: The **split** operation enforces an upper bound on the number of items owns by a non free peer. Also, as shown in Theorem 2, after a split, $lb \leq |p.own|, |q.own| \leq ub$.

Merge Operation:
CONDITION: When a non free peer p owns $\leq lb$ entries, it either tries to get some entries from its neighbors (successor or predecessor), or gives up the entries to its predecessor and becomes free. The former case, called the *redistribute* happens if p has a neighbor on the ring (successor or predecessor) which owns at least $\frac{ub}{2}$ entries. The latter, called *merge* happens when neither of p's neighbours have at least $\frac{ub}{2}$ entries.
ALGORITHM:

1: **if** $|p.own| > lb$ **then**
2: return;
3: **end if**
4: $p_1 = \texttt{succ}(p)$;
5: $p_2 = \texttt{pred}(p)$;
6: $op =$ MERGE;
7: **if** $|p_1.own| \geq \frac{ub}{2}$ **then**
8: $q = p_1$; $op =$ REDISTRIBUTE;
9: **else**
10: **if** $|p_2.own| \geq \frac{ub}{2}$ **then**
11: $q = p_2$; $op =$ REDISTRIBUTE;
12: **else**
13: $q = p_2$; $op =$ MERGE;
14: **end if**
15: **end if**
16: **if** $op =$ REDISTRIBUTE **then**
17: transfer $\frac{ub}{4} - \frac{lb}{2}$ entries from $q.own$ to $p.own$;
18: redistribute new $p.own$ amongst $\mathcal{H}(p)$;
19: redistribute new $q.own$ amongst $\mathcal{H}(q)$;
20: **else**
21: $q.own = q.own + p.own$;
22: $p.leaveRing()$;
23: $\mathcal{H}(q) = \mathcal{H}(q) \cup \mathcal{H}(p) \cup \{p\}$;
24: **end if**

PURPOSE: The **merge** operation ensures that the number of items owned by a peer does not fall below lb. Also, as shown in Theorem 2, after a redistribute, $lb \leq |p.own|, |q.own| \leq ub$, and after a merge, $lb \leq |q.own| \leq ub$.

Usurp Operation:
CONDITION: Consider a non free peer $p_1 \in \mathcal{N}$, and a free peer $q \in \mathcal{F}$, $\psi(q) = p_2$. Given a constant δ, if $|p_1.resp| \geq 2\sqrt{1+\delta}|q.resp|$, then p_1 can $usurp$ the free peer q and set q as its helper peer.
ALGORITHM:

1: find least loaded free peer q ($\psi(q) = p_2$);
2: **if** $|p_1.resp| \geq 2(\sqrt{1+\delta})|q.resp|$ **then**
3: $p_1.setHelperPeer(q)$;
4: redistribute $p_1.own$ amongst its new set of helper;
5: redistribute $p_2.own$ amongst remaining helpers;
6: **end if**

PURPOSE: The first two operations only talk about bounding entries owned by non free peers. However, load imbalance is defined as the imbalance in $p.resp$ for all $p \in \mathcal{P}$. This operation bounds the load imbalance between two peers p and q, where at least one of them is a free peer, by $2\sqrt{1+\delta}$ (see Theorem 2). Note that $p.own$ does not change for any non free peer p due to $usurp$ operations.

The algorithm EXTLOADBALANCE performs the appropriate operation when the appropriate condition is met. We assume that in the $usurp$ operation, a non free peer can easily find the least loaded free peer and updates are reflected immediate. This could be implemented by building an index on the load of free peers. We do not elaborate on this aspect here.

Let us revisit now the issue of setting the upper and lower bounds. In the basic scheme discussed in the previous section, we set $lb = \texttt{sf}$ and $ub = 2\texttt{sf}$. Since we needed free peers to exist whenever a peer needed to split, we set $\texttt{sf} = \lceil N/n \rceil$, where $n = |\mathcal{P}|$ and N is the total number of entries in the system; i.e., $n(d-1) < N \leq nd$, for some integer d implies $\texttt{sf} = \texttt{d}$. If due to inserts, $N > nd$, sf should be updated to $(d+1)$ and if due to deletes, $N \leq n(d-1)$, sf should be updated to $(d-1)$.

In algorithm EXTLOADBALANCE, for efficiency reasons (see Theorem 3), we need $ub \geq 2\texttt{sf}$. Henceforth, we assume that $lb = d$ and $ub = (2+\epsilon)d$ for some constant $\epsilon > 0$. In EXTLOADBALANCE, if due to inserts, $N > nd$, we set sf to $(d+1)$. But, for efficiency reasons (see Theorem 3), we do not update sf to $(d-1)$ until $N \leq n(d-1-\gamma)$, for some positive constant $\gamma < \frac{1}{2}$. Hence, $lb = d$ implies $n(d-1-\gamma) \leq N \leq nd$. This change, however, maintains the property that there are free peers available whenever a peer wants to split (see Lemma 1).

Lemma 1 *Whenever algorithm* EXTLOADBALANCE *performs the $split$ operation, there exists a free peer in the system.*

Proof. Let $lb = d$. Then $ub \geq 2d$. Suppose there are no free peers when algorithm EXTLOADBALANCE wants to perform a split operation. Since there is a peer owning ub entries, the total number of entries in the system, N, is at least $(n-1)d + 2d$; i.e., $N > nd$. However, ub and lb are set such that, if $lb = d$, then $n(d-1-\gamma) \leq N \leq nd$. This leads to the required contradiction. ■

Definition 2 (Valid Configuration) *Let n be the number of peers and N the number of entries in the system. Let constants $\gamma \leq \frac{1}{2}$, lb and ub be such that $\frac{ub}{lb} \geq 2$ and $lb = d$ implies $n(d-1-\gamma) \leq N \leq nd$, A configuration described by the tuple $(\mathcal{R}, \rho, \psi)$ is said to be a* valid configuration*, if for some positive constant δ, the ownership and responsibility assignments completely defined by the configuration satisfy:*

1. Ownership Validity*: $lb < |p.own| < ub$ for all non free peers $p \in \mathcal{N}$, and*

2. Responsibility Validity*: if there are free peers in the system and $q \in \mathcal{F}$ is the free peer responsible for the least number of items, any peer p is such that $|p.resp| \leq 2\sqrt{1+\delta}|q.resp|$.*

We can bound the maximum number of helper peers assigned to a non free peer to a constant κ_h^{valid} in a valid configuration (see Lemma 2).

Lemma 2 *In a valid configuration, the number of helper peers assigned to a non free peers is at most* $\kappa_h^{valid} = 4\frac{ub}{lb}\sqrt{1+\delta} - 1$.

Proof. Consider a non free peer p owning $\ell = |p.own|$ entries and having h helpers. p and each of its helpers are responsible for $\frac{\ell}{1+h}$ entries. Since the current configuration is valid, we have that any other peer q in the system is responsible for at most $\frac{\ell}{1+h}2\sqrt{1+\delta}$ entries. Let N denote the total number of items in the system. We know that

$$N \geq n(lb - 1 - \gamma)$$

Also, from the above discussion,

$$N \leq (n-h-1)\frac{\ell}{1+h}2\sqrt{1+\delta} + \ell$$

From the above two equations, we have

$$h < \frac{\ell}{lb-2}2\sqrt{1+\delta} \leq 4\frac{ub}{lb}\sqrt{1+\delta}$$

■

We are now ready to state the main theorems regarding the correctness, load imbalance and the efficiency of EXTLOADBALANCE.

Theorem 1 (Correctness) *Starting from any initial* valid configuration *defined by $(\mathcal{R}_0, \rho_0, \psi_0)$, for every insert or delete operation, algorithm* EXTLOADBALANCE *performs a finite sequence of* split, merge *and* usurp *operations and results in a valid configuration.*

Corollary 1.1 *Starting from a valid configuration,* EXTLOADBALANCE *fixes a violation due to an insert or a delete by performing at most one $split$ or one $merge$ operation followed by a constant number of $usurp$ operations.*

Theorem 2 (Load Imbalance) *The algorithm always returns to a valid configuration and hence the load imbalance is at most* $\max \frac{ub}{lb}, 2\sqrt{1+\delta}$

Our cost model is very similar to the one used in Ganesan et al. [11]. There are three major components to the cost involved:

- Data Movement: we model this cost as being linear in the number of entries moved from one peer to the other.
- Distributing entries amongst helper peers: this happens whenever the set of entries owned by a peer p or the set of helpers $\mathcal{H}(p)$ changes. $|p.own|$, the number of items which have to be distributed amongst the helpers, is a very conservative estimate on the number of entries moved around.
- Load Information: Our algorithm requires non-local information about the least loaded free peer. We assume that this comes at zero cost.

Under this cost model, we prove the following efficieny result.

Theorem 3 (Efficiency) *Starting from an initial configuration defined by $(\mathcal{R}_0, \rho_0, \psi_0)$, for every sequence σ of item inserts and deletes, if $\frac{ub}{lb} \geq 2(1+\epsilon)$ for some $\epsilon > 0$, the sequence of* split, merge *and* usurp *operations performed by algorithm* EXTLOADBALANCE *for any prefix of σ is such that the amortized cost of an insert or a delete operation in that prefix of σ is a constant.*

Before we go on to prove the theorems, we define a $potential$ Φ associated with every configuration $(\mathcal{R}, \rho, \psi)$. We will use this potential to prove the above stated theorems. Henceforth, we will assume that $lb = d$ and $ub = (2+\epsilon)d$ for some ϵ.

Definition 3 (Potential) *We define for each configuration $(\mathcal{R}, \rho, \psi)$ a potential $\Phi = \Phi_o + \Phi_r$, where Φ_o and Φ_r are defined as follows:*

The $OwnershipPotential$ $\Phi_o = \sum_{p \in \mathcal{P}} \phi_o(p)$, where for some constant c_o,

$$\phi_o(p) = \begin{cases} 0 & p \notin \mathcal{N} \text{ (free peer)} \\ \frac{c_o}{d}(l_0 - |p.own|)^2 & d \leq |p.own| \leq l_0 \\ 0 & l_0 \leq |p.own| \leq u_0 \\ \frac{c_o}{d}(|p.own| - u_0)^2 & u_0 \leq |p.own| \leq (2+\epsilon)d \end{cases}$$

$$l_0 = (1 + \frac{\epsilon}{4})d$$

$$u_0 = (2 + \frac{3\epsilon}{4})d$$

The $ResponsibilityPotential$ $\Phi_r = \sum_{q \in \mathcal{P}} \phi_r(q)$, where for some constant c_r

$$\phi_r(q) = \frac{c_r}{d}(|q.resp|)^2$$

We now quantify the change in the ownership and responsibility potentials on an insert, a delete and respectively a single load balancing operation. Define $\Delta_{op}\Phi$ (and respectively $\Delta_{op}\Phi_o$ and $\Delta_{op}\Phi_r$) to be the decrease in potential Φ (Φ_o and Φ_r, respectively) due to one of the above stated operations.

Insert: Due to an insert, an entry is inserted into $p.own$ for some p and inserted into $q.resp$ for some $q \in \mathcal{H}(p) \cup \{p\}$. Hence, $\phi_r(p)$ will increase and $\phi_o(p)$ will increase when

$u_0 \leq |p.own| \leq (2+\epsilon)d$. Hence, the minimum decrease in Φ occurs when both $\phi_r(q)$ and $\phi_o(p)$ increase and this decrease is

$$\begin{aligned}
\Delta_{ins}\Phi_o &= \frac{c_o}{d}(|p.own| - u_0)^2 - \frac{c_o}{d}(|p.own| + 1 - u_0)^2 \\
&= -\frac{c_o}{d}(2|p.own| + 1 - 2u_0) \\
&\geq -\frac{c_o}{d}(2(2+\epsilon)d + 1 - 2(2+\frac{3\epsilon}{4})d) \\
&\geq -\frac{c_o}{d}(\frac{\epsilon}{2}d + 1) \\
&\geq -\frac{c_0\epsilon}{2} \\
\Delta_{ins}\Phi_r &= \frac{c_r}{d}(|q.resp|)^2 - \frac{c_r}{d}(|q.resp| + 1)^2 \\
&= -\frac{c_r}{d}(2|q.resp| + 1) \\
&\geq -\frac{c_r}{d}(2(2+\epsilon)d + 1) \\
&\geq -2(2+\epsilon)c_r
\end{aligned}$$

$$\Delta_{ins}\Phi \geq -\frac{c_o\epsilon}{2} - 2(2+\epsilon)c_r \quad (1)$$

Delete: Due to a delete, an entry is deleted from $p.own$ for some non free peer p and deleted from $q.resp$ for some $q \in \mathcal{H}(p) \cup \{p\}$. Like in the insert case, the minimum decrease in Φ occurs when both $\phi_r(q)$ and $\phi_o(p)$ increase. Here, $\phi_o(p)$ increases when $d \leq |p.own| \leq l_0$.

$$\begin{aligned}
\Delta_{del}\Phi_o &= \frac{c_o}{d}(l_0 - |p.own|)^2 - \frac{c_o}{d}(l_0 - (|p.own| - 1))^2 \\
&= -\frac{c_o}{d}(2l_0 - 2|p.own| + 1) \\
&\geq -\frac{c_o}{d}(2(1+\frac{\epsilon}{4})d + 1 - 2d) \\
&\geq -\frac{c_o}{d}(\frac{\epsilon}{2}d + 1) \\
&\geq -\frac{c_0\epsilon}{2} \\
\Delta_{del}\Phi_r &= \frac{c_r}{d}(|q.resp|)^2 - \frac{c_r}{d}(|q.resp| + 1)^2 \\
&\geq -2(2+\epsilon)c_r
\end{aligned}$$

$$\Delta_{del}\Phi \geq -\frac{c_o\epsilon}{2} - 2(2+\epsilon)c_r \quad (2)$$

Split: First let us look at the decrease in the ownership potential $\Delta_{split}\Phi_o$. During a split, a peer p owning $|p.own| = (2+\epsilon)d$ entries, gives half of its entries to a free peer q. After the split, both p and q own $(1+\frac{\epsilon}{2})d$ items. Hence, the final ownership potentials of p and q are 0. Also, the initial ownership potential of q is 0 since before the split q was a free peer.

$$\Delta_{split}\Phi_o \geq \frac{c_o}{d}((2+\epsilon)d - u_0)^2 = c_o(\frac{\epsilon}{4})^2 d$$

Next, consider the change in the responsibility potential. When $\mathcal{H}(p) \neq \emptyset$, q is chosen from $\mathcal{H}(p)$ and the helper peers are distributed amongst p and q evenly. In this case, the responsibilities change only when the number of helpers apart from q ($|\mathcal{H}(p) \setminus \{q\}|$) is odd, say $2h+1$. This is because the $(1+\frac{\epsilon}{2})d$ entries in p and q are distributed amongst $h+1$ and $h+2$ peers respectively. In this case the decrease in Φ_r would be

$$\begin{aligned}
\Delta_{split}\Phi_r &= (2h+3)\frac{c_r}{d}\left(\frac{(2+\epsilon)d}{2h+3}\right)^2 \\
&\quad -(h+1)\frac{c_r}{d}\left(\frac{\left(1+\frac{\epsilon}{2}\right)d}{h+1}\right)^2 \\
&\quad -(h+2)\frac{c_r}{d}\left(\frac{\left(1+\frac{\epsilon}{2}\right)d}{h+2}\right)^2 \\
&\geq \frac{c_r}{d}\left(\frac{((2+\epsilon)d)^2}{4(h+2)} - \frac{((2+\epsilon)d)^2}{4(h+1)}\right) \\
&= c_r(2+\epsilon)^2 d\left(\frac{1}{4(h+2)} - \frac{1}{4(h+1)}\right) \\
&= -\frac{c_r(2+\epsilon)^2}{4(h+1)(h+2)}d \\
\Delta_{split}\Phi_r &\geq -\frac{c_r(2+\epsilon)^2}{8}d
\end{aligned}$$

When p does not have any associated helper peers, say p takes over p_2's helper q. Let, $h = |\mathcal{H}(p_2)|$, $\ell = |p_2.own|$. We have,

$$\begin{aligned}
\Delta_{split}\phi_r(p) &= \frac{c_r}{d}\left(((2+\epsilon)d)^2 - \left(\left(1+\frac{\epsilon}{2}\right)d\right)^2\right) \\
\Delta_{split}\phi_r(q) &= \frac{c_r}{d}\left(\left(\frac{\ell}{1+h}\right)^2 - \left(\left(1+\frac{\epsilon}{2}\right)d\right)^2\right)
\end{aligned}$$

$\forall q_2 (\neq q) \in \mathcal{H}(p) \cup \{p_2\}$,

$$\begin{aligned}
\Delta_{split}\phi_r(q_2) &= \frac{c_r}{d}\left(\left(\frac{\ell}{1+h}\right)^2 - \left(\frac{\ell}{h}\right)^2\right) \\
\Delta_{split}\Phi_r &= \frac{c_r}{d}\left(\frac{1}{2}((2+\epsilon)d)^2 + \frac{\ell^2}{1+h} - \frac{\ell^2}{h}\right) \\
&= \frac{c_r}{d}\left(\frac{((2+\epsilon)d)^2}{2} - \frac{\ell^2}{h(h+1)}\right) \\
&\geq \frac{c_r}{d}\left(\frac{((2+\epsilon)d)^2}{2} - \frac{((2+\epsilon)d)^2}{1\cdot 2}\right) \\
&\geq 0
\end{aligned}$$

Hence the minimum decrease in the potential due to a split is

$$\Delta_{split}\Phi \geq c_o(\frac{\epsilon}{4})^2 d \quad (3)$$

Merge: REDISTRIBUTE: Let us first quantify the decrease in ownership potential. A peer p_1 which owns $|p_1.own| = d$ entries gets $\frac{\epsilon}{4}d$ entries from p_2 which owns at least $(1+\frac{\epsilon}{2})d$ entries. Before the redistribute, $(1+\frac{\epsilon}{2})d \leq |p_2.own| \leq$

$(2+\epsilon)d$. Hence, after the redistribute, we have $(1+\frac{\epsilon}{4})d \leq |p_2.own| \leq (2+\frac{3\epsilon}{4})d$. Hence, p_2's final ownership potential is 0. Also, p_1's final ownership potential is 0 since it will now have $(1+\frac{\epsilon}{4})d$ entries. Since, p_2's initial potential could have been 0, the minimum decrease in ownership potential is given by

$$\Delta_{redist}\Phi_o \geq \frac{c_o}{d}(l_0-d)^2 = c_o(\frac{\epsilon}{4})^2 d$$

For the change in the responsibility potential, let $|\mathcal{H}(p_1)| = h_1$ and $|\mathcal{H}(p_2)| = h_2$. Let $|p_2.own| = \ell$. We have,

$$\begin{aligned}
&\forall q_1 \in \mathcal{H}(p_1)\cup\{p_1\},\\
&\Delta_{redist}\phi_r(q_1) = \frac{c_r}{d}\left(\left(\frac{d}{1+h_1}\right)^2 - \left(\frac{\left(1+\frac{\epsilon}{4}\right)d}{1+h_1}\right)^2\right)\\
&\forall q_2 \in \mathcal{H}(p_2)\cup\{p_2\},\\
&\Delta_{redist}\phi_r(q_2) = \frac{c_r}{d}\left(\left(\frac{\ell}{1+h_2}\right)^2 - \left(\frac{\ell-\frac{\epsilon}{4}d}{1+h_2}\right)^2\right)
\end{aligned}$$

$$\begin{aligned}
\Delta_{redist}\Phi_r &= -\frac{c_r}{1+h_1}\left(\frac{\epsilon}{2}d + \left(\frac{\epsilon}{4}\right)^2 d\right)\\
&\quad + \frac{c_r}{1+h_2}\left(\frac{\epsilon}{2}\ell - \left(\frac{\epsilon}{4}\right)^2 d\right)\\
&\geq \frac{c_r\epsilon}{2}\left(\frac{\left(1+\frac{\epsilon}{2}\right)d}{1+h_2} - \frac{d}{1+h_1}\right)\\
&\quad - \frac{c_r\epsilon^2 d}{16}\left(\frac{h_2+h_1}{(1+h_1)(1+h_2)}\right)\\
&\geq -c_r d\left(\frac{\epsilon}{2}\left(1-\frac{1+\frac{\epsilon}{2}}{\kappa_h}\right) + \frac{\epsilon^2}{16}\left(1+\frac{1}{\kappa_h}\right)\right)\\
&\geq -c_r d\left(\frac{\epsilon}{2}+\frac{\epsilon^2}{8}\right)
\end{aligned}$$

Hence the maximum decrease in potential is

$$\Delta_{redist}\Phi \geq c_o d\frac{\epsilon^2}{16} - c_r d\left(\frac{\epsilon}{2}+\frac{\epsilon^2}{8}\right) \qquad (4)$$

MERGE: Considering the ownership potential, a peer p_1 which has $|p_1.own| = d$ entries, gives up all its entries to p_2 and becomes free. Hence, p_1's final ownership potential is 0. p_2 has at most $(1+\frac{\epsilon}{2})d$ entries before the merge. Hence, p_2's final ownership potential is also 0. Hence, the decrease in ownership potential is at least p_1's initial potential, which is

$$\Delta_{merge}\Phi_o \geq \frac{c_o}{d}(l_0-d)^2 = c_o\left(\frac{\epsilon}{4}\right)^2 d$$

Considering the responsibility potential, let $|\mathcal{H}(p_1)| = h_1$ and $|\mathcal{H}(p_2)| = h_2$. Let $|p_2.own| = \ell \leq \left(1+\frac{\epsilon}{2}\right)d$. In the merge, p_1 gives up all its entries to p_2, becomes free, and all the free peers in $\{p_1\}\cup\mathcal{H}(p_1)$ become p_2's helpers. Note that there might be too many helpers for p_2 after the merge, but a sufficient number of $usurp$ operations will reduce the number of free peers to below κ_h.

$$\begin{aligned}
&\forall q_1 \in \mathcal{H}(p_1)\cup\{p_1\},\\
&\Delta_{merge}\phi_r(q_1) = \frac{c_r}{d}\left(\left(\frac{d}{1+h_1}\right)^2 - \left(\frac{\ell+d}{2+h_1+h_2}\right)^2\right)\\
&\forall q_2 \in \mathcal{H}(p_2)\cup\{p_2\},\\
&\Delta_{merge}\phi_r(q_2) = \frac{c_r}{d}\left(\left(\frac{\ell}{1+h_2}\right)^2 - \left(\frac{\ell+d}{2+h_1+h_2}\right)^2\right)
\end{aligned}$$

$$\begin{aligned}
\Delta_{merge}\Phi_r &= \frac{c_r}{d}\left(\frac{d^2}{1+h_1} + \frac{\ell^2}{1+h_2} - \frac{(\ell+d)^2}{2+h_1+h_2}\right)\\
&\geq -c_r\left(\frac{(\ell+d)^2}{2+h_1+h_2}\right)\\
&\geq -2c_r\left(1+\frac{\epsilon}{4}\right)d
\end{aligned}$$

Hence, the maximum decrease in potential is

$$\Delta_{merge}\Phi \geq c_o\left(\frac{\epsilon}{4}\right)^2 d - 2c_r\left(1+\frac{\epsilon}{4}\right)d \qquad (5)$$

Usurp: In the usurp operation, the ownership mappings do not change. Hence the decrease in ownership potential due to an usurp operation is 0.

The responsibility potential, however, decreases in this operation. Let p_1 be a non free peers with $|p_1.own| = \ell_1$ and $|\mathcal{H}(p_1)| = h_1$. Let q be the free peer usurped by p_1. If $p_2 = \psi(q)$, $|p_2.own| = \ell_2$ and $|\mathcal{H}(p_2)| = h_2 \geq 1$, then $\frac{\ell_1}{1+h_1} \geq 2\sqrt{1+\delta}\frac{\ell_2}{1+h_2}$. Let h_{max} be the maximum number of free peers assigned to a non free peer in the current configuration. Note that the current configuration is not a valid configuration and hence the maximum number of free peers might exceed κ_h^{valid}.

$$\begin{aligned}
&\forall q_1 \in \mathcal{H}(p_1)\cup\{p_1\},\\
&\Delta_{usurp}\phi_r(q_1) = \frac{c_r}{d}\left(\left(\frac{\ell_1}{1+h_1}\right)^2 - \left(\frac{\ell_1}{2+h_1}\right)^2\right)\\
&\forall q_2 (\neq q) \in \mathcal{H}(p_2)\cup\{p_2\},\\
&\Delta_{merge}\phi_r(q_2) = \frac{c_r}{d}\left(\left(\frac{\ell_2}{1+h_2}\right)^2 - \left(\frac{\ell_2}{h_2}\right)^2\right)\\
&\Delta_{merge}\phi_r(q) = \frac{c_r}{d}\left(\left(\frac{\ell_2}{1+h_2}\right)^2 - \left(\frac{\ell_1}{2+h_1}\right)^2\right)
\end{aligned}$$

$$\begin{aligned}
\Delta_{merge}\Phi_r &= \frac{c_r}{d}\left(\frac{\ell_1^2}{1+h_1} - \frac{\ell_1^2}{2+h_2} + \frac{\ell_2^2}{1+h_2} - \frac{\ell_2^2}{h_2}\right) \\
&= \frac{c_r}{d}\left(\frac{\ell_1^2}{(1+h_1)(2+h_1)} - \frac{\ell_2^2}{h_2(1+h_2)}\right) \\
&\geq \frac{c_r}{d}\left(\frac{\ell_1^2}{2(1+h_1)^2} - \frac{2\ell_2^2}{(1+h_2)^2}\right) \\
&\geq \frac{c_r}{d}\left(\frac{2\sqrt{1+\delta}\ell_2^2}{(1+h_2)^2} - \frac{2\ell_2^2}{(1+h_2)^2}\right) \\
&\geq \frac{2c_r\delta}{(1+h_{max})^2}\frac{d^2}{d} \\
&\geq \frac{2c_r\delta}{(1+h_{max})^2}d
\end{aligned}$$

$$\Delta_{usurp}\Phi \geq \frac{2c_r\delta}{(1+h_{max})^2}d \quad (6)$$

Proof. of Theorem 1: Consider a valid initial configuration which satisfies the *ownership* and *responsibility validity* conditions. If an insert violates the ownership constraint, there is one peer p which violates the constraint by owning ub entries. A split is performed which results in p owning $(1+\frac{\epsilon}{2})d$ entries and a free peer q being added to the ring with the same number of entries. Hence, the new configuration satisfies the ownership constraint.

If a delete causes an ownership violation, there is one peer p which violates the constraint by owning lb entries. A merge operation is performed. In the case of a redistribute, p_2 owning $(1+\frac{\epsilon}{2})d \leq |p.own| \leq (2+\epsilon)d$ entries gives $\frac{\epsilon}{4}$ to p_1. Thus now p_1 owns $(1+\frac{\epsilon}{4})d$ entries and p_2 owns $(1+\frac{\epsilon}{4})d \leq (2+\frac{3\epsilon}{4})d$ entries, satisfying the ownership constraints. In the case of a merge, p_1 gives up its d entries to p_2 and becomes free. p_2 initially has at most $(1+\frac{\epsilon}{2})d$ entries and hence addition of d more entries will not violate the ownership constraints.

We showed that violation of an ownership constraint can be fixed using one split or merge operation. However, this operation might lead to violation of a responsibility constraint. This is fixed by using usurp operations. We prove that the number of usurp operations required is finite by using a simple potential argument. Recall the responsibility potential Φ_r. Starting from a valid configuration, a configuration resulting from an insert or a delete satisfies the following modified ownership constraint: for all $p \in \mathcal{P}$, $d \leq |p.own| \leq (2+\epsilon)d$. Hence the potential Φ_r is linear in d. The potential of the resulting valid configuration is positive. From equation 6, the decrease in Φ_r due to an usurp operation is greater than 0. Hence, in a finite number of usurp operations, we reach a valid configuration. ■

Lemma 3 *In any configuration attained during the execution of* EXTLOADBALANCE*, the number of helper peers assigned to a non free peers is at most* $\kappa_h \leq (4+\epsilon)\kappa_h^{valid}$*, where* $\frac{ub}{lb} = (2+\epsilon)$.

Proof. We first prove that any configuration attained during the execution of EXTLOADBALANCE satisfies a variant of Property 2 of a valid configuration, namely

2' If there are free peers in the system and $q \in \mathcal{F}$ is the free peer responsible for the least number of items, any peer p is such that $|p.resp| \leq (2+\frac{\epsilon}{2})2\sqrt{1+\delta}|q.resp|$.

Proving the above result would imply that the maximum number of free peers assigned to a non free peer is at most $(2+\frac{\epsilon}{2})\kappa_h^{valid}$ using the same arguments as in Lemma 2.

Let us step through the execution of EXTLOADBALANCE on a sequence of inserts and deletes. Let ℓ_{min} be the minimum number of entries a free peer q is responsible for and ℓ_{max} be the maximum number of entries a non free peer is responsible for.

- Starting from a valid configuration with an imbalance of at most $2\sqrt{1+\delta}$, an insert or a delete cannot result in a configuration with a greater imbalance than $4\sqrt{1+\delta}$, since only one item is added or removed.
- An insert or a delete could be followed by a split or a merge operation respectively. Let us consider an insert followed by a split. Say peer p had $(2+\epsilon)d-1$ entries in the valid configuration and an insert violated the ownership constraint at p. If p has a helper q, p splits with q and distributes the helpers in $\mathcal{H}(p)\backslash\{q\}$ between themselves. Let $\mathcal{H}(p) = h$. In the initial configuration, all peers in $\mathcal{H}(p)\cup\{p\}$ are responsible for $\frac{(2+\epsilon)d-1}{1+h}$. If h were odd, then finally both p and q would share the items and helpers equally and the responsibility does not change. However, if h were even, say $2m$, then finally, p would be responsible for $\frac{(1+\frac{\epsilon}{2})d}{m}$ entries and q and its helpers would be responsible for $\frac{(1+\frac{\epsilon}{2})d}{1+m}$. However, the final responsibilities are off from the initial responsibilities by at most a factor of 2 and hence even in the worst case the imbalance is not more than $4\sqrt{1+\delta}$.
- Consider a delete followed by a redistribute. Initially, p_1 is responsible for $\frac{d+1}{1+h_1}$ entries and p_2 is responsible for at least $\frac{(1+\frac{\epsilon}{2})d}{1+h_2}$ entries. Finally, p_1 is responsible for $\frac{(1+\frac{\epsilon}{4})d}{1+h_1}$ and p_2 is responsible for at least $\frac{(1+\frac{\epsilon}{4})d}{1+h_1}$. In the case of p_1 and its helpers, the load could be imbalanced by an additional factor of at most $(1+\frac{\epsilon}{4})$ while in the case of p_2 the load could be imbalanced by an additional factor of 2.
- Consider a delete followed by a merge. In this case, p_1 owns ℓ_1 entries with h_1 helpers, p_2 owns ℓ_2 entries with h_2 helpers. Finally, p_1 and p_2 and each of the helpers is responsible for $\frac{\ell_1+\ell_2-1}{2+h_1+h_2}$. Since $\frac{\ell_1}{1+h_1}$ and $\frac{\ell_2}{1+h_2}$ did not violate the responsibility condition, $\frac{\ell_1+\ell_2-1}{2+h_1+h_2}$ can violate the condition by at most an additional factor of 2.

- For the usurp operation, we can show that the imbalance after an usurp operation is not greater than the imbalance before the operation. Let p_1 usurp p_2's helper. Let initially p_1 and p_2 own ℓ_1 and ℓ_2 entries and have h_1 and h_2 helpers respectively. For the usurp to occur, $\frac{\ell_1}{1+h_1} \geq 2\sqrt{1+\delta}\frac{\ell_2}{1+h_2}$. We show that $\frac{\ell_2}{1+h_2} \leq \frac{\ell_1}{2+h_1}, \frac{\ell_2}{h_2} \leq \frac{\ell_1}{1+h_1}$. Hence the result.

$$\begin{aligned}
\frac{\ell_1}{2+h_1} &\geq \frac{\ell_1}{2(1+h+1)} \\
&\geq \frac{\sqrt{1+\delta}\ell_2}{1+h_2} \\
&> \frac{\ell_2}{1+h_2} \\
\frac{\ell_2}{h_2} &\leq \frac{2\ell_2}{1+h_2} \\
&\leq \frac{\ell_1}{\sqrt{1+\delta}(1+h_1)} \\
&< \frac{\ell_1}{1+h_1}
\end{aligned}$$

Hence, we see that the additional imbalance is not more than a factor of $\max 2, (1+\frac{\epsilon}{4})$ and hence the imbalance is at worst $(2+\frac{\epsilon}{2})2\sqrt{1+\delta}$. ■

Proof. of Corollary 1.1: From the proof of the Theorem 1, we know that the ownership violation caused by an insert or a delete can be fixed in exactly one split or merge.

Since κ_h, the maximum number of free peers assigned to a non free peer (Lemma 3), is a constant, $\Delta_{usurp}\Phi$ is a fraction of d. From the proof of Theorem 1, we know that any attainable configuration has a potential linear in d. Since every usurp operation decreases this potential by a fraction of d, and the potential of the resulting valid configuration is positive, the number of usurp operations is at most a constant. ■

Proof. of Theorem 2: From Theorem 1, we know that the algorithm always returns to a valid configuration. Also in a valid configuration, the imbalance between free and non free peers is at most $2\sqrt{1+\delta}$ and the imbalance between non free peers is at most $\frac{ub}{lb}$. Hence the result follows. ■

Proof. of Theorem 3: First note that the increase in the potential Φ on an insert or a delete is at most $\frac{\epsilon}{2}c_o + 2(2+\epsilon)c_r$, which is a constant given c_o and c_r. If we can set the constants c_o and c_r such that the minimum decrese in Φ is greater than the maximum cost of a load balancing opertation, we are done proving the amortized constant cost of an insert or a delete.

The cost of a **split** operation is at most $(3+\frac{3\epsilon}{2})d$. In the case when p, the splitting peer has a helper q, the cost is contributed by the transfer of $(1+\frac{\epsilon}{2})d$ entries and the rearranging of entries amongst p's and q's helpers. Hence the total cost is at most $(3+\frac{3\epsilon}{2})d$. When p does not have a helper, p takes away q from some other non free peer p_2. Here the cost involved transfer of entries from p to q and the rearranging of entries amongst p_2's remaining helpers. Hence, the cost is at most $(3+\frac{3\epsilon}{2})d$. Hence we need c_o and c_r such that

$$c_o\left(\frac{\epsilon}{4}\right)^2 d \quad \geq \quad \left(3+\frac{3\epsilon}{2}\right)d \tag{7}$$

The cost of the REDISTRIBUTE case in a **merge** operation, is at most $(3+\frac{5\epsilon}{4})d$. The cost involves transfer of $\frac{\epsilon}{4}$ entries and the redistribution of the final set of entries owned by p_1 and p_2 amongst their helpers.

$$c_o d\frac{\epsilon^2}{16} - c_r d\left(\frac{\epsilon}{2}+\frac{\epsilon^2}{8}\right) \quad \geq \quad \left(3+\frac{5\epsilon}{4}\right)d \tag{8}$$

The cost of a MERGE is at most $(3+\frac{\epsilon}{2})d$, since the cost only involves transfer of d entries to the more loaded peer and redistribution of at most $(2+\frac{\epsilon}{2})d$ entries amongst the new set of helper peers. Hence,

$$c_o\left(\frac{\epsilon}{4}\right)^2 d - 2c_r\left(1+\frac{\epsilon}{4}\right)d \quad \geq \quad \left(3+\frac{\epsilon}{2}\right)d \tag{9}$$

Finally, the **usurp** operation costs $\ell_1+\ell_2 \leq 2(2+\epsilon)d$, where the two non free peers involved own ℓ_1 and ℓ_2 entries respectively. The cost arises due to the redistribution amongst the new set of helpers. Hence,

$$\frac{2c_r\delta}{\kappa_h^2}d \quad \geq \quad 2(2+\epsilon)d \tag{10}$$

Solving equations 7, 8, 9, 10, we get

$$\begin{aligned}
c_o\frac{\epsilon^2}{16} &\geq c_r\left(1+\epsilon+\frac{\epsilon}{4}\right)+\left(3+\frac{3\epsilon}{2}\right) \\
c_r &\geq \frac{(2+\epsilon)\kappa_h^2}{\delta}
\end{aligned}$$

By setting the constants c_r and c_o to values as shown above, we can prove that the amotized cost of inserts and deletes is a constant when d does not change.

We still need to consider the case when lb changes. lb changes either due to N becoming greater than nd due to inserts, in which case $lb \leftarrow d+1$ or due to N becoming lesser than $n(d-1-\gamma)$, $\gamma < \frac{1}{2}$, in which case $lb \leftarrow d-1$. Due to the change in d the potential might increase. Note that the change affects only the ownership potential and not the responsibility potential.

- *Increase in Φ due to lb changing from d to $d+1$:*
  In this case, $l_0 = (1+\frac{\epsilon}{4})d$ increases by $(1+\frac{\epsilon}{4})$. Hence for all the peers p owning $|p.own| = \ell \leq l_0$ entries,

the potential increases and the increase is

$$\begin{aligned} inc\ \Phi &\geq n\frac{c_o}{d}\left(\left(l_0+\left(1+\frac{\epsilon}{4}\right)-\ell\right)^2-(l_0-\ell)^2\right) \\ &= n\frac{c_o}{d}\left(2(l_0-\ell)+\left(1+\frac{\epsilon}{4}\right)\right)\left(1+\frac{\epsilon}{4}\right) \\ &\leq n\frac{c_o}{d}\left(2d\frac{\epsilon}{4}\right)\left(1+\frac{\epsilon}{4}\right) \\ inc\ \Phi &\leq nc_o\frac{\epsilon}{2}\left(1+\frac{\epsilon}{4}\right) \end{aligned}$$

- *Increase in Φ due to lb changing from d to $d-1$:* Similarly in this case, u_0 decreases by $(1+\frac{\epsilon}{4})$. Hence for all the peers p owning $|p.own| = \ell \geq u_0$ entries, the potential increases and the increase is

$$inc\ \Phi \leq nc_o\frac{\epsilon}{2}\left(1+\frac{\epsilon}{4}\right)$$

If we are able to show that the number of insert/delete operations between two consecutive changes in lb is linear in n, we can charge the increase in potential as a constant overhead cost to each insert/delete operation and thus prove that the amortized cost is a constant even with changes in lb.

Let us count the number of steps between two consecutive changes in lb.

- lb changes from $d-1$ to d and then to $d+1$: In this case, change from $d-1$ to d happens when $N > n(d-1)$ and the change from d to $d+1$ happens when $N > nd$. Hence there are at least n inserts between the changes.
- lb changes from $d+1$ to d and then to $d-1$: In this case, change from $d+1$ to d happens when $N \leq n(d-\gamma)$ and the change from d to $d-1$ happens when $N \leq n(d-1-\gamma)$. Hence there are at least n deletes between the changes.
- lb changes from $d+1$ to d and then back to $d+1$: In this case, change from $d+1$ to d happens when $N \leq n(d-\gamma)$ and the change from d to $d+1$ happens when $N > nd$. Hence there are at least $n\gamma$ inserts between the changes.
- lb changes from $d-1$ to d and then to $d-1$: In this case, change from $d+1$ to d happens when $N > n(d-1)$ and the change from d to $d-1$ happens when $N \leq n(d-1-\gamma)$. Hence there are at least $n\gamma$ deletes between the changes.

Thus there are at least $n\gamma$ inserts/deletes between two consecutive changes in lb. Hence by charging each insert/delete an extra constant cost of

$$\frac{c_o\epsilon}{2\gamma}\left(1+\frac{\epsilon}{4}\right)$$

we can pay for the operations caused by the change in lb also. ■

4. P-Ring Content Router

The goal of our Content Router is to efficiently route messages to peers in a given range. The main challenge in designing a Content Router for range queries is to handle skewed distributions. Since the search key values distribution can be skewed, the ranges assigned to the peers may not be of equal length. Consequently, index structures that assume uniform data distribution in the indexing domain such as Chord [33] and Pastry [30] cannot be applied in this case. Recently, some P2P indexing structures that can handle skewed distributions have been proposed [8, 2, 15], but these structures either provide only probabilistic search guarantees [2, 15], or do not provide search guarantees [8] even in a stable system.

The existing work on distributed B+-trees is not directly applicable in a massively distributed system like ours. To the best of our knowledge, all such index structures [17, 20] try to maintain a globally consistent B+-tree by *replicating* the nodes of the tree across different processors. The consistency of the replicated nodes is then maintained using *primary copy* replication. Relying on primary copy replication creates both scalability (load/resource requirements on primary copy) and availability (failure of primary copy) problems, and is clearly not a solution for a large-scale P2P systems with thousands of peers.

We devise a new content router called *Hierarchical Ring* (or short, HR) that can handle highly skewed data distributions. In the following sections, we describe the content router and the routing and maintenance algorithms. We then analytically bound the search performance in a stable system and under very heavily skewed insertion patterns. We also experimentally evaluate the content router in the perspective of our architecture.

4.1. Hierarchical Ring

The HR Content Router is based on the simple idea of constructing a hierarchy of rings.

Let d be an integer ¿ 1, called the 'order' of HR. At the lowest level, level 1, peer p maintains a list of the first d successors on the ring. Using the successors, a message could always be forwarded to the last successor in the list that does not overshoot the target "skipping" up to d-1 peers at a time. Consider the ring in Figure 6, where peer p_1 is responsible for the range $(5, 10]$, peer p_2 is responsible for range $(10, 15]$ and so on and assume that d=2. Each peer knows its successor on the ring: $\texttt{succ}(p_1) = p_2$, $\texttt{succ}(p_2) = p_3$, ..., $\texttt{succ}(p_5) = p_1$. At level 1 in the Content Router, each peer maintains a list of 2 successors, as shown. Suppose p_1 needs to route a message to a peer with value 20. In this case, p_1 will route the message to p_3 and p_3 will forward the message to p_5, the final destination.

At level 2, we again maintain a list of d successors. However, a successor at level 2 corresponds to the dth successor at level 1. Note that using these successors, a message

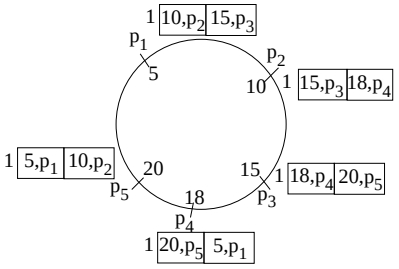

Figure 6. HR Level 1

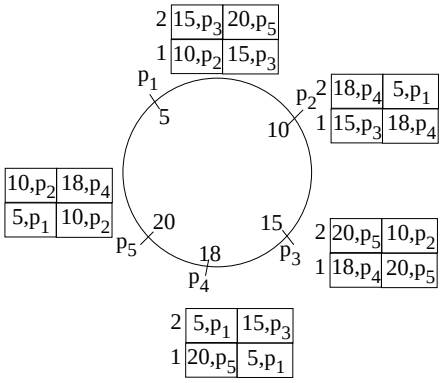

Figure 7. HR Levels 1 and 2

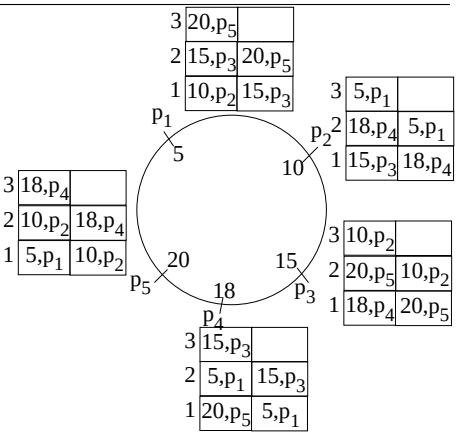

Figure 8. HR Levels 1, 2, and 3

could always be routed to the last successor in the list that does not overshoot the target, "skipping" up to $d^2 - 1$ peers at a time. Figure 7 shows the content of level 2 nodes at each peer in the ring. Suppose that p_1 needs to route a message to a peer with value 20. p_1 will route the message directly to p_5 (the final destination), using the list at level 2. The procedure of defining the successor at level $l + 1$ and creating a list of level $l + 1$ successors is iterated until no more levels can be created. In Figure 8, for peer p_1 for example, note that $\texttt{succ}_3(p_5) = p_4$, which overshoots p_1, so no more levels can be constructed for p_1.

An important observation about this index structure is that we are conceptually indexing "positions" in the ring (i.e. at level l, a peer p has pointers to peers that are d^l $peers$ away) instead of values, which allows the structure to handle skewed data distributions.

Formally, the data structure for a HR of order d is a doubly indexed array $node[level][position]$, where $1 \leq level \leq numLevels$ and $1 \leq position \leq d$. The HR is defined to be consistent if and only if at each peer p:

- $p.node[1][1] = \texttt{succ}(p)$
- $p.node[1][j+1] = \texttt{succ}(p.node[1][j])$, $1 \leq j < d$
- $p.node[l+1][1] = p.node[l][d]$,
- $p.node[l+1][j+1] = p.node[l+1][j].node[l+1][1]$, $1 \leq l < numLevels$, $1 \leq j < d$
- The successor at $numLevels$ of the last peer in the list at $numLevels$ level "wraps" around, so all the peers are indeed indexed: $p.node[numLevels].lastPeer.node[numLevels][1] \in [p, p.node[numLevels].lastPeer)$

From this definition, it is easy to see that a consistent HR of order d, has only $\lceil \log_d(P) \rceil$ levels, and the space requirement for the Content Router component at each peer is $O(d \cdot \log_d(P))$, where P is the number of peers in the fault tolerant ring.

4.2. Maintenance

Peer failures and insertions disrupt the consistency of the HR. We have a remarkably simple *Stabilization Process* that runs periodically at each peer and repairs the inconsistencies in the HR. The algorithm guarantees that the HR structure eventually becomes fully consistent after any pattern of concurrent insertions and deletions, as long as the peers remain connected at the fault-tolerant ring level.

The algorithm executed periodically by the Stabilization Process is shown in Algorithm 5. The algorithm loops from the lowest level to the top-most level of the HR until the highest (root) level is reached (as indicated by the boolean variable *root*). Since the height of the HR data structure could actually change, we update the height ($p.numLevels$) at the end of the function.

Algorithm 6 describes the Stabilization Process within each level of the HR data structure at a peer. The key observation is that each peer needs only local information to compute its own successor at each level. Thus, each peer relies on other peers to repair their own successor at each level. When a peer p stabilizes a level, it contacts its successor at that level and asks for its entries at the corresponding level. Peer p replaces its own entries with the received entries and inserts its successor as the first entry in the index node (lines 2 and 3). The INSERT procedure, apart from inserting the specified entry at the beginning of the list at given level, it also ensures that no more than d entries are in the list and none of the entries in the list overshoots p (the list does not wrap around). Line 4 checks whether this level should be the last level in the HR. This is the case if all the peers in the system are already covered. If this level is not the root level, the stabilization procedure computes the successor at the higher level (line 7) and returns.

4.3. Routing

The Content Router component supports the `sendReceive(msg,range)` primitive. We as-

Algorithm 5 : p.Stabilize()

1: $i = 1$;
2: **repeat**
3: root=p.StabilizeLevel(i);
4: $i++$;
5: **until** (root)
6: $p.numLevels = i - 1$;

Algorithm 6 : p.StabilizeLevel(int i)

1: $succEntry = p.node[i][1]$;
2: $p.node[i]$ = $succEntry.node[i]$;
3: INSERT($i, succEntry$);
4: **if** $p.node[i].lastPeer.node[i][1] \in [p, p.node[i].lastPeer)$ **then**
5: return `true`
6: **else**
7: $p.node[i+1][1]$ = $p.node[i][d]$;
8: return `false`;
9: **end if**

sume that each routing request originates at some peer p in the P2P system. For simplicity of presentation, we assume that the `range` has the form $(lb, ub]$.

The routing procedure takes as input the lower-bound (lb) and the upper-bound (ub) of the range in the request, the message that needs to be routed, and address of the peer where the request was originated; the pseudo-code of the algorithm is shown in Algorithm 7. We denoted by `rangeMin(p)` the low end value of $p.range$. The routing procedure at each peer selects the farthest away pointer that does not overshoot lb and forwards the request to that peer. Once the algorithm reaches the lowest level of the HR, it traverses the successor list until the value of a peer exceeds ub (lines 8-9). Note that every node which is responsible for a part of $(lb, ub]$ is visited during the traversal along the ring. At the end of the range scan, a $SearchDoneMessage$ is sent to the peer that originated the search (line 11).

Example: Consider a routing request for the range $(18, 25]$ that is issued at peer p_1 in Figure 8. The routing algorithm first determines the highest HR level in p_1 that contains an entry whose value is between 5 (value stored in p_1) and 18 (the lower bound of the range query). In the current example, this corresponds to the first entry at the second level of p_1's HR nodes, which points to peer p_3 with value 15. The routing request is hence forwarded to p_3. p_3 follows a similar protocol, and forwards the request to p_4 (which appears as the first entry in the first level in p_3's HR nodes). Since p_4 is responsible for items that fall within the required range, p_4 processes the routed message and returns the results to the originator p_1 (line 6). Since the successor of p_4, p_5, might store items in the $(18, 25]$ range, the request is also forwarded to p_5. p_5 processes the request and sends the results

Algorithm 7 : p.`routeHandler`(lb, up, msg, $originator$)

1: // find maximum level that contains an
2: // entry that does not overshoot lb.
3: find the maximum level l such that $\exists\ j > 0$ such that $p.node[l][j].iValue \in$ (`rangeMin(p)`, `lb`].
4: **if** no such level exists **then**
5: //handle the message and send the reply
6: send($p.handleMessage(msg)$, $originator$);
7: **if** `rangeMin(succ(p))` $\in$ (`rangeMin(p)`, `ub`] **then**
8: // if successor satisfies search criterion
9: send(`Route`(lb,ub,msg,$originator$,$requestType$), `succ`(p));
10: **else**
11: send(`RoutingDoneMessage`,$originator$);
12: **end if**
13: **else**
14: find maximum k such that $p.node[l][k].iValue \in$ (`rangeMin(p)`, `lb`];
15: send(`Route`((lb,ub,msg,$originator$), $p.node[l][k].peer$));
16: **end if**

to p_1. The search terminates at p_5 as the value of its successor (5) does not fall within the query range.

In a consistent state, the routing procedure will go down one level in the HR every time a routing message is forwarded to a different peer. This guarantees that we need at most $\lceil \log_d(P) \rceil$ steps, if the HR is consistent. If a HR is inconsistent, however, the routing cost may be more than $\lceil \log_d(P) \rceil$. Note that even if the HR is inconsistent, it can still route requests by using the nodes to the maximum extent possible, and then sequentially scanning along the ring. In Section 5.2, we experimentally show that the search performance of HRs does not degrade much even when the index is temporarily inconsistent.

It is important to note that in a P2P system we cannot guarantee that every route request terminates. For example, a peer p could crash in the middle of processing a request, in which case the originator of the request would have to time out and try the routing request again. This model is similar to that used in most other P2P systems [30, 33, 29].

4.4. Properties of Hierarchical Ring

In this section we describe some of the formal properties of the Hierarchical Ring.

Definition We define a stabilization unit (`su`) to be the time needed to run the `StabilizeLevel` procedure at some level in all peers.

Theorem 4 (Stabilization time) *Given that at time t there are P peers in the system and the fault tolerant ring is connected and the stabilization procedure starts running periodically at each peer, at time $t + (d-1)\lceil \log_d(P) \rceil$* `su` *the HR is consistent with respect to the P peers.*

Proof sketch: The stabilization starts at time t by stabilizing level 1 which already has the correct first entry (since the fault-tolerant ring is connected). After at most one stabilization unit, each peer finds out about its successor's successor and so on. After running the `StabilizeLevel` procedure $d-1$ times at level 1, each peer has level 1 in HR and the first entry in level 2 consistent . Since there are $\lceil \log_d(P) \rceil$ levels, after $(d-1)\lceil \log_d(P) \rceil$ stabilization units, the HR is consistent with respect to the P peers. ■

Theorem 5 (Search performance in stable state) *In a stable system of P peers with a consistent Hierarchical Ring data structure of order d, equality queries take at most $\lceil \log_d(P) \rceil$ hops.*

Theorem 6 (Search performance during insertions) *If we have a stable system with a consistent HR of order d data structure and we start inserting peers at the rate r peers/stabilization unit, then equality queries take at most $\lceil \log_d(P) \rceil + 2r(d-1)\lceil \log_d(P) \rceil$ hops, where P is the current number of peers in the system.*

Proof sketch: Let t_0 be the initial time and P_0 be the number of peers in the system at time t_0. For every $i > 0$ we define t_i to be $t_{i-1} + (d-1)\lceil \log_d(P_{i-1}) \rceil \cdot \mathtt{su}$ and P_i to be the number of peers in the system at time t_i. In the following, we call an "old" peer to be a peer that can be reached in at most $\lceil \log_d(P) \rceil$ hops using the HR. If a peer is not "old", we call it "new". At any time point, the worst case search cost for equality queries is $\lceil \log_d(P) \rceil + x$, where $\lceil \log_d(P) \rceil$ is the maximum number of hops using the HR to find an old peer and x is the number of new peers. x is also the maximum number of hops to be executed using the successor pointers to find any one of the new x peers (the worst case is when all new peers are successors in the ring). We will show by induction on time that the number of new peers in the system at any time cannot be higher than $2r(d-1)\lceil \log_d(P) \rceil$.

As the base induction step we prove that at any time point in the interval $[t_0, t_1]$ there are no more than $2r(d-1)\lceil \log_d(P) \rceil$ new peers and at time t_1 there are no more than $rd\lceil \log_d(P) \rceil$ new peers. From hypothesis, at t_0 the HR is consistent, so there are no new peers. At the insertion rate of r peers/$\mathtt{su}$, at any time point in $[t_0, t_1]$, the maximum number of peers inserted is $r(d-1)\lceil \log_d(P_0) \rceil$, which is smaller than $r(d-1)\lceil \log_d(P) \rceil$. This proves both statements of the base induction step.

We prove now that if the maximum number of new peers at time t_i is $rd\lceil \log_d(P) \rceil$, than, at any time point in $[t_i, t_{i+1}]$ the maximum number of new peers is $2r(d-1)\lceil \log_d(P) \rceil$ and the maximum number of new peers at time t_{i+1} is $r(d-1)\lceil \log_d(P) \rceil$, where $i \geq 1$. The maximum number of peers inserted between t_i and t_{i+1} is $r(d-1)\lceil \log_d(P_i) \rceil$ which is smaller than $r(d-1)\lceil \log_d(P) \rceil$. From the induction hypothesis, at time t_i there were at most $r(d-1)\lceil \log_d(P) \rceil$ new peers. Between t_i and t_{i+1}, some old peers can become new and new peers can become old, due to changes in the HR structure. However, the total number of entries in the HR structure does not decrease, so the number of old peers becoming new cannot be higher than the number of new peers becoming old. Out of the peers in the system at time t_i, at most $r(d-1)\lceil \log_d(P) \rceil$ of them are new at any time between t_i and t_{i+1}. Adding the peers inserted since t_i we get that at any time point in $[t_i, t_{i+1}]$ the maximum number of new peers is $2r(d-1)\lceil \log_d(P) \rceil$. From Theorem 4, at time t_{i+1}, all the peers existing in the system at time t_i are integrated into the HR structure. This means that all peers existing at time t_i are/became old peers at time t_{i+1}, which leaves the maximum number of new peers at time t_{i+1} to be at most $r(d-1)\lceil \log_d(P) \rceil$ (the peers inserted between t_i and t_{i+1}).

From induction it follows that at any time, the maximum number of new peers is no more than $2r(d-1)\lceil \log_d(P) \rceil$, which means that equality queries take at most $\lceil \log_d(P) \rceil + 2r(d-1)\lceil \log_d(P) \rceil$ hops. ■.

5. Experimental Evaluation

We focus on two main aspects in our experimental evaluation. First, we evaluate the performance of the P-Ring Data Store. As a baseline, we compare it with the hash-based Chord Data Store, which does not support range queries. Second, we evaluate the performance of the P-Ring Content Router, and compare it with Skip Graphs. We also consider the interaction between the two components in the presence of peer insertions and deletions (system "churn").

5.1. Experimental Setup

We developed a simulator in C++ to evaluate the index structures. We implemented the P-Ring Data Store and the Chord Data Store, and the P-Ring Content Router and SkipGraphs. Since SkipGraphs was originally designed for only a single item per peer, we extended it to use the P-Ring Data Store so that it could scale to multiple items per peer. For all the approaches, we used the same Fault Tolerant Ring [33] and the Replication Manager [7] so that we can isolate the differences amongst the Data Stores and Content Routers.

We used three main performance metrics. The first metric is the *index message cost*, which is the number of messages per second (in simulator time units) required for maintaining the index. The second metric is the *index bandwidth cost*, which is the number of bytes per second required for maintaining the index (since not all messages are of the same size). The third metric is the *search cost*, which is the number of messages required to evaluate a query. In our experiments, we calculate the search cost by averaging the number of messages required to search for a random value in the system starting from 100 random peers. Since the main variable component in the cost of range queries is finding the data item with the smallest qualifying value (retrieving the other values has a fixed cost of traversing the

relevant leaf values), we only measure the cost of finding the first entry for range queries. This also enables us to compare against the performance of Chord for equality queries.

In our experiments we varied the following parameters: *InsertionRate* (similarly, *DeletionRate*) is the rate of insertions (deletions) into the system. *ItemInsertionPattern* (similarly, *ItemDeletionPattern*, specifies the skew in the data values inserted (deleted) into the system. A value of *ip* for this parameter means that all insertions are localized within a fraction *ip* of the search key space (default is 1). *NumPeers* is the number of peers in the system (default is 2000). *PeerIDRate* is the rate of peer insertions and failures in the system; insertions and failures are equally likely. For each the of experiments below, we vary one parameter and we use the default values for the rest. We first evaluate the Data Store and Content Router components separately in a stable system configuration (without peer failures); we then investigate the effect of peer failures.

5.2. Experimental Results: Data Store

We now study the performance of P-Ring Data Store. Note that the performance of the Data Store depends on the performance of the Content Router (when searching for free peers). To isolate these effects as much as possible, we fix the P-Ring Content Router to have orders 2 and 10 for this set of experiments (we investigate different orders in subsequent sections). As a baseline for comparison, we use the Chord Data Store, which is efficient due to hashing, but does not support range queries.

5.2.1. Varying Item Insertion Rate Figure 9 shows the index message cost as a result of varying *InsertionRate*. The message cost increases linearly with *InsertionRate* because each item insertion requires a search message to locate the peer that should store the item. The message cost increases faster for the P-Ring Data Store than for Chord because the P-Ring additionally needs to periodically split and merge due to item skew. In contrast, the Chord datastore is more efficient because it simply hashes data items to peers and does not have any item redistribution overhead. This difference quantifies the additional overhead of supporting range queries (using the P-Ring datastore) as opposed to simple equality queries (using the Chord datastore). Finally, we note that that the message cost for the P-Ring Data Store decreases as we use a Content Router of higher order - this is because the search for free peers becomes more efficient with higher order Content Routers. The graph showing the index bandwidth cost is similar and is not shown. We also obtained similar results by varying *ItemDeletionPattern*.

5.2.2. Varying Item Insertion Pattern Figure 10 shows the index message cost as a result of varying *ItemInsertionPattern* (recall that 0 corresponds to highly skewed distribution, while 1 corresponds to a uniform distribution). For the Chord Data Store, as expected, we do not observe any significant variation in message cost. The surprising aspect, however, is that the message cost also remains relatively stable for P-Ring Data Store even under highly skewed distributions. This suggests that the P-Ring Data Store effectively manages item skew by splitting and merging as required. The graph showing the index bandwidth cost is similar, and we also obtained similar results by varying *ItemDeletionPattern*.

5.3. Experimental Results: Content Router

We now investigate the performance of the P-Ring Content Router, and compare it with SkipGraphs and Chord. Since Chord cannot directly handle range queries, we artifically specify queries over the *hash value* for Chord (which is its best-case scenario) so that we can compare Content Routers in terms of performance.

5.3.1. Varying Number of Peers Figure 11 shows the search cost when varying the number of peers. As expected, the search cost increases logarithmically with the number of peers (note the logarithmic scale on the x-axis) for all the Content Routers. However, the search costs for the different Content Routers varies significantly. In particular, SkipGraphs has significantly worse search cost because the index structure of order d has search performance $O(d \times \log_d(N))$ (where N is the number of peers in the system). In contrast, Chord has search cost $O(\log_2(N))$ and a P-Ring of order d has search cost $O(\log_d(N))$. For this reason, the P-Ring of order 10 significantly outperforms the other index structures due to the large base of the logarithm.

5.3.2. Varying Order Figure 12 shows the effect of varying the order d of P-Ring on the search cost. As expected, the search cost is $O(\log_d(N))$, where N is the number of peers in the system (recall the default is $N = 2000$). Figures 13 and 14 show how the index message cost and index bandwidth cost, respectively, vary with order. The index message cost steadily decreases with order because there are fewer levels in the Content Router that need to be stabilized (recall that the number of levels in a Content Router of order d is $\log_d(N)$). However, the index bandwidth cost decreases slightly and then increases because, at higher orders, a lot more information has to be transferred during index stabilization. Specifically, each stabilization in a Content Router of order d has to transfer $O(d)$ information (the entries at one level). Hence, the total bandwidth requirement is $O(d \cdot \log_d(N))$, which is consistent with the experimental results. This shows the tradeoff between index stabilization and search cost - a higher value of d improves search but increases bandwidth requirements.

5.4. Experimental Results: System Churn

Figure 15 shows the effect of peer insertions and failures on index performance, for 4 insertions/failures per second (the results with other rates is similar). The basic tradeoff is between search cost and index bandwidth cost. When the Content Router is stabilized at a high rate, this leads to a high bandwidth cost due to many stabilization messages,

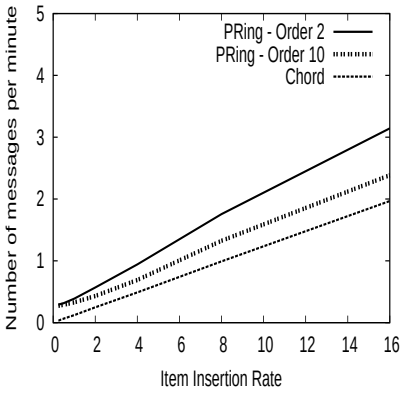

Figure 9. Item Insertion Rate

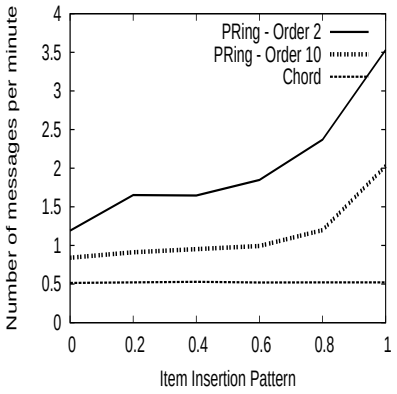

Figure 10. Item Insertion Pattern

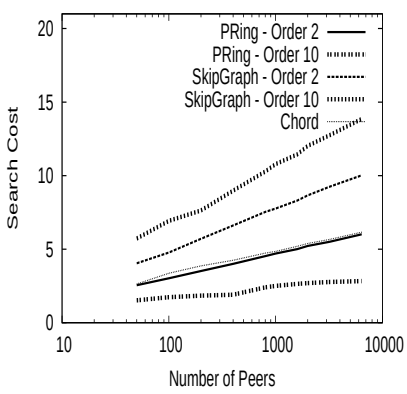

Figure 11. Number of Peers

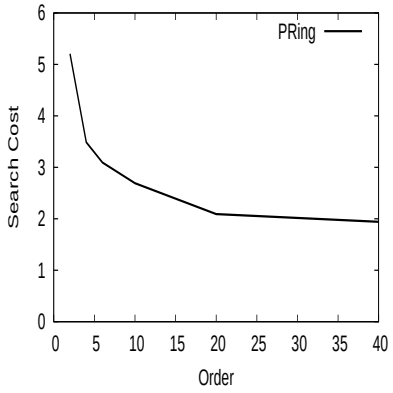

Figure 12. Varying Order

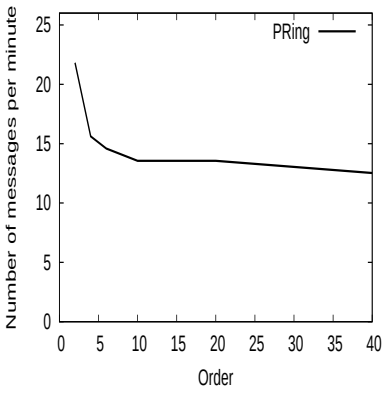

Figure 13. Varying Order

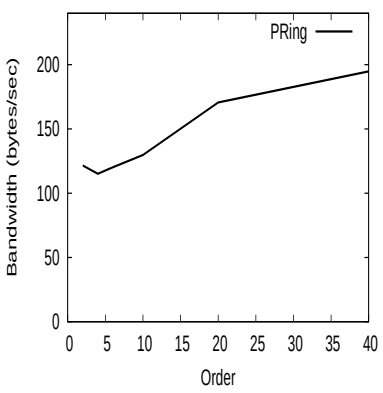

Figure 14. Varying Order

but a low search cost since the Content Router is more consistent. On the other hand, when the Content Router is stabilized very slowly, the bandwidth cost decreases but the search cost increases. Note that this also affects the Data Store performance because the Data Store uses the Content Router for inserting/deleting items and finding free peers.

As shown in Figure 15, the P-Ring Content Router always dominates SkipGraphs due to its superior search performance. Chord outperforms P-Ring of order 2 because Chord does not have the overhead of dealing with splits and merges during system churn. However, P-Ring of order 10 offers a better search cost, albeit at a higher bandwidth cost, while still supporting range queries. We also obtained similar results for search cost vs. index message cost, and hence the results are not shown.

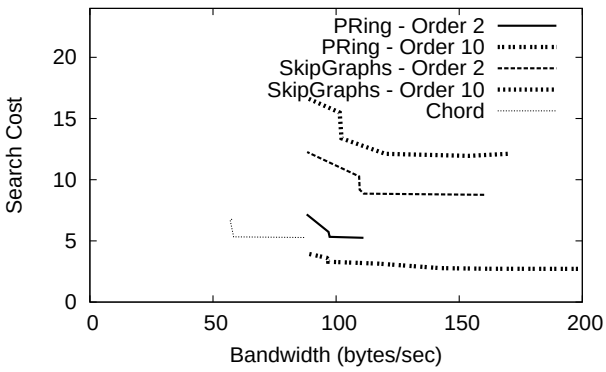

Figure 15. Search vs Bandwidth Cost

6. Related Work

There has been recent work on P2P data management issues like schema mediation [3, 18, 32], query processing [28], and the evaluation of complex queries such as joins [14, 32]. However, none of these approaches address the issue of supporting range queries efficiently.

CAN [29], Chord [33], Pastry [30] and Tapestry [36] implement distributed hash tables to provide efficient lookup of a given key value. Since a hash function destroys the ordering in the key value space, these structures cannot process range queries efficiently. Approaches to the lookup

problem based on prefix matching/tries [1, 9, 30] cannot be used to solve the range query problem for arbitrary numerical attributes such as floating point numbers. Other approaches to the lookup problem include [12, 26, 35]. Techniques for efficient keyword search are presented in [6, 27, 34]. None of these systems support range queries.

There has been work on developing distributed index structures [16, 20, 23, 24]. However, most of these techniques maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability problems when dealing with thousands of peers. In contrast, the P-Ring data structure is designed to be resilient to extended failures of arbitrary peers. The DRT [21] and dPi-tree [25] maintain replicas lazily, but these schemes are not designed for peers that can leave the system, which makes them inadequate in a P2P environment.

Gupta et al. [13] present a technique for computing range queries in P2P systems using order-preserving hash functions. Since the hash function scrambles the ordering in the value space, their system can only provide approximate answers to range queries (as opposed to the exact answers provided by P-trees). Aspnes et al. propose Skip graphs [2], a randomized structure based on skip lists, which supports range queries. Unlike P-Ring, they only provide probabilistic guarantees even when the index is fully consistent. Daskos et al. [8] present another scheme for answering range queries, but the performance of their system depends on certain heuristics for insertions. Their proposed heuristics do not offer any performance guarantees and thus, unlike P-Ring, their search performance can be linear in the worst case even after their index becomes fully consistent. Galanis et al. [10] describe an index structure for locating XML documents in a P2P system, but this index structure does not provide any provable guarantees on size and performance, and is not designed for a highly volatile environment. Sahin et al. [31] propose a caching scheme to help answer range queries, but their scheme does not provide any performance guarantees for range queries which were not previously asked.

In concurrent work, Ganesan et al. [11] propose a load balancing scheme for data items where they prove a bound of 4.24 for storage imbalance with constant amortized insertion and deletion cost. The P-Ring data store achieves a better storage balance with a factor of $2+\epsilon$ with the same amortized insertion and deletions cost. Additionally, we also propose a new content router, the Hierarchical Ring.

Finally, the P-Ring evolved from the P-Tree index structure [4]. Unlike the P-Tree, the P-Ring supports multiple items per peer and offers provable search guarantees not only in a stable state but also during insertions and deletions.

7. Conclusion

We have introduced P-Ring, a novel fault-tolerant P2P index structure that efficiently supports *both* equality and range queries in a dynamic P2P environment. P-Ring effectively balances data items among peers even in the presence of skewed data insertions and deletions and provides provable guarantees on search performance. Our experimental evaluation shows that P-Ring outperforms existing index structures, sometimes even for equality queries, and that it maintains its excellent search performance with low maintenance costs in a dynamic P2P system.

References

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, 2001.

[2] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.

[3] P. Bernstein et al. Data management for peer-to-peer computing: A vision. In *WebDB*, 2002.

[4] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB*, 2004.

[5] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. An indexing framework for peer-to-peer systems. In *WWW (poster)*, 2004.

[6] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer networks. In *ICDCS*, 2002.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[8] A. Daskos, S. Ghandeharizadeh, and X. An. Peper: A distributed range addressing space for p2p systems. In *DBISP2P*, 2003.

[9] M. J. Freedman and R. Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *IPTPS*, 2002.

[10] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[11] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[12] P. Ganesan, Q. Sun, and H. Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *INFOCOM*, 2003.

[13] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[14] M. Harren et al. Complex queries in dht-based peer-to-peer networks. In *IPTPS*, 2002.

[15] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, 2003.

[16] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *IPPS*, 1992.

[17] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD*, 1993.

[18] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: a new flavor of federated query processing for db2. In *SIGMOD*, 2002.

[19] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *FOCS*, 2003.

[20] P. A. Krishna and T. Johnson. Index replication in a distributed b-tree. In *COMAD*, 1994.

[21] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *SIGMOD*, 1994.

[22] C. Lagoze and H. V. de Sompel. The open archive initiative: Building a low-barrier interoperability framework. In *JCDL*, 2001.

[23] W. Litwin, M.-A. Neimat, and D. A. Schneider. Lh* - linear hashing for distributed files. In *SIGMOD*, 1993.

[24] W. Litwin, M.-A. Neimat, and D. A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In *VLDB*, 1994.

[25] D. B. Lomet. Replicated indexes for distributed data. In *PDIS*, 1996.

[26] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC*, 2002.

[27] Neurogrid - http://www.neurogrid.net.

[28] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[29] S. Ratnasamy et al. A scalable content-addressable network. In *SIGCOMM*, 2001.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[31] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[32] W. Siong Ng et al. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, 2003.

[33] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[34] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS*, 2002.

[35] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *ICDE*, 2003.

[36] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. In *Technical Report, U.C.Berkeley*, 2001.